How to Write-All Efficiently Even with Contaminated Memory

Alex A. Shvartsman

Technical Report No. CS-92-09

February 1992

How to Write-All efficiently even with contaminated memory

Alex A. Shvartsman[*]

January 31, 1991

Abstract

The problem of *Write-All*—using P-processors write 1's into all locations of an array of size N, where $P \leq N$— has been used as the basic building block for constructing efficient and fault-tolerant parallel algorithms. All previous *Write-All* solutions use $\Omega(P)$ auxiliary shared memory and assume that this memory is cleared or initialized to some known value. When *Write-All* building blocks are used in polylogarithmic parallel time algorithms (e.g., to compute prefix sums or list ranking) auxiliary memory initialization cannot be amortized over the computation. Thus, assuming clear memory is a very strong precondition and for *Write-All* itself raises a legitimate "chicken-or-egg" objection.

In this note, using a deterministic bootstrapping and balancing argument, we show how to *Write-All* when auxiliary memory is contaminated with arbitrary values. For any dynamic pattern of fail-stop, no-restart errors on a CRCW PRAM with at least one surviving processor, our new algorithm writes all 1's using $O(N+P\log^3 N/(\log\log^2 N))$ work, ***without any initialization assumption***. This technique can be combined with any *Write-All* algorithm to yield efficient simulations of any PRAM and even optimal simulations given processor slack. It can also be used with restartable fail-stop processor simulations. In addition, we show that for the parallel prefix computation it is possible to improve on the best deterministic simulations to date: by a factor of $\log N$ when the memory is clear and by a factor of $\log\log N$ when the memory is contaminated.

[*]*Department of Computer Science, Brown University, PO Box 1910, Providence, RI 02912, USA.* **Email:** **aas@cs.brown.edu**, and *Digital Equipment Corporation, LKG2-2/T2, 550 King Street, Littleton, MA 01460, USA.* **This research was supported by Digital Equipment Corp. and ONR grant N00014-91-J-1613.**

1 Introduction

Related work and motivation:

The study of fault-tolerant and asynchronous parallel algorithms for the parallel random access machine (PRAM [8]) has attracted a fair amount of recent attention. Several efficient algorithms have been designed for PRAMs that are subject to stop-failures or to processor delays, where this processor behavior is determined by adversaries of varying strengths. For example: asynchronous PRAMs are the subject of [1, 4, 5, 6, 9, 13, 18, 19], and fault-prone PRAMs are studied in [4, 11, 12, 13, 20]. The motivation of this research area is to bridge the gap between realizable parallel computers and the PRAM, with its unrealistic features of broad bandwidth memory access, processor synchrony and freedom from faults. Our work is in the area of asynchronous and fault-prone models, but we do use broad bandwidth access to shared memory as a means of providing redundancy when encountering faults. For a detailed discussion of the general model used and how it can be realized see [4].

Here, we reexamine the key problem of *Write-All* and remove a strong initialization assumption that has been used in all its previous solutions. *Write-All* was formulated in [11] in order to show that it is possible to combine efficiency and fault tolerance in the presence of arbitrary dynamic fail-stop PRAM processor errors. Its solutions have been used to compile PRAM algorithms for architectures where asynchrony or processor failures are present. It can be formulated as follows:

Using P-processors write 1's into all locations of an array of size N, where $P \leq N$.

Write-All captures the computational progress that can be naturally accomplished in unit time by a PRAM (when $P = N$). In the presence of asynchrony or failures, efficient solutions to *Write-All* (increasing the fault-free work by polylogarithmic factors only) are non-obvious. Note that, in all existing solutions it does not matter what is the initial state of the size N array. For example we assume it is all 0's in [11, 4, 20], but the algorithms would work even if the N locations were initialized using arbitrary 0's and 1's. A much more important assumption in all previous *Write-All* solutions was the initial state of additional auxiliary memory used (typically of $\Omega(P)$ size). The basic assumption has been that:

The $\Omega(P)$ auxiliary shared memory is cleared or initialized to some known value.

In theory, this is a natural, even if unstated assumption, for PRAMs [8] and RAMs (cf., Turing Machine auxiliary tapes are initially blank). However, given the definition of *Write-All* this dependence on clear space raises a legitimate "chicken-or-egg" objection. In practice, memory locations typically contain unpredictable values, and processes that need to use large blocks of memory cannot assume that it is cleared or is initialized to a known value. In fact operating systems usually provide explicit services that allocate clear memory, e.g., **calloc()** in standard C libraries. Such allocation is predictably much more time consuming, even in the absence of failures.

It is easy to construct simple *Write-All* algorithms that do not assume clear shared memory, but they appear to use quadratic work. If the overall computation involves many steps, one can perhaps afford an expensive initialization phase and amortize its cost over subsequent efficient steps. Unfortnately, when *Write-All* building blocks are used in very fast (i.e., polylogarithmic parallel time) algorithms (e.g., to compute prefix sums or list ranking) auxiliary memory initialization cannot be amortized over the computation. Fortunately, we show that there is a way around this dilemma:

> *We present Write-All algorithms and algorithm simulations that do not require that the auxiliary memory is cleared prior to the computation.*

Algorithms in the setting studied in the present paper have some similarities with the notion of a *self-stabilizing system* introduced by Dijkstra in [7]. Paraphrasing [7], a system is self-stabilizing if and only if, regardless of the initial state the system can always make a state transition into another state, and the system is guaranteed to find itself in a legitimate state after a finite number of transitions. Our computations using initially contaminated memory can be viewed as self-stabilizing with respect to the state of shared memory. In order to describe our technical contributions we must now review the state-of-the-art of the algorithmics of *Write-All.*

For the worst case on-line stop-failures without restarts, Kanellakis and Shvartsman [11] gave an efficient (within a $\log^2$ factor) algorithm for *Write-All* (algorithm W) and other key problems using an iterated *Write-All* paradigm. This paradigm was then employed independently by Kedem et al. [12] and Shvartsman [20] to extend the results of [11] to arbitrary PRAM algorithms. In addition, Kedem et al. [12] analyzed the expected behavior of several solutions to *Write-All* using a random failure model. Shvartsman [20] presented a deterministic optimal $O(N)$ work execution of PRAM algorithms subject to worst case failures by exploiting parallel slackness with $P \leq N/\log^2 N$. A simple randomized *Write-All* algorithm that can be used for simulating arbitrary PRAM algorithms on an asynchronous PRAM is presented by Martel et al. in [18]; this simulation has very good expected performance when the adversary is off-line. Kedem et al. [13] have shown an $\Omega(N \log N)$ lower bound on work, for any deterministic *Write-All* solution. In addition, they have shown an $O(N\frac{\log^2 N}{\log\log N})$ deterministic work upper bound on *Write-All.* Their upper bound is based on a variation of algorithm W, and it has been shown by Martel [16] that the same upper bound applies to algorithm W [11].

For the worst case on-line stop-failures with restarts there has also been some progress. A parallel model where processors are subject to failures and restarts is examined by Buss et al. in [4]. This framework generalized previous models of robust parallel computations and in it *Write-All* has a subquadratic $O(N^{1.59})$ work solution. Martel et al. [17] presented several randomized solutions for list ranking and sorting that have very efficient expected work when the scheduling adversary is off-line. An efficient randomized solution for the *Write-All* problem was developed by Anderson and Woll in [1] for the asynchronous parallel model. They have also showed an existence proof for an algorithm achieving work $O(N^{1+\varepsilon})$ for any $\varepsilon > 0$. General synchronous PRAM simulations are impossible using *bounded* resources on asynchronous PRAMS because of the impossibility result shown by Herlihy [10]. However the algorithms in [1] can be used with the restartable fail-stop model defined by Buss et al. [4] (which restricts asynchrony). We will take advantage of this since general simulations are possible in that model.

Contributions:

We eliminate the assumption that any amount of clear initial memory is available for the fail-stop and fail-stop restartable algorithms. We develop *deterministic* fault-tolerant algorithms that can be used to simulate PRAMs using contaminated memory, i.e., when the shared memory not containing the input is initially in an arbitrary and possibly illegal state. We also improve on the state-of-the-art robust prefix sums computations. More specifically:

1. In the no-restart fail-stop parallel model, any N-processor PRAM algorithm that runs in time τ can be deterministically simulated using $O(N)$ contaminated memory on P fail-stop processors with work $O(N + P\log^3 N/(\log\log N)^2 + \tau \cdot P\log^2 N/\log\log N)$ for $1 \leq P \leq N$.

   This simulation has an optimal range of processors, i.e., the work of the simulation is asymptotically equal to the work of the simulated non-fault-tolerant algorithm.
2. In the restartable fail-stop model, any N-processor PRAM algorithm that runs in time τ can be simulated using $O(N)$ contaminated memory on $P = N$ restartable fail-stop processors with $S = O(\tau \cdot N^{1+\epsilon})$.
3. For the parallel prefix computation it is possible to improve on the oblivious simulations of non-fault-tolerant algorithm (e.g., the ones we get by using [12, 20] with conventional algorithms). In order to compute the prefix sums of N values using N processors, at least $\log N/\log\log N$ parallel steps are required [2, 15], and the known algorithms require at least $\log N$ steps. Therefore an oblivious simulation of a known prefix algorithm will require simulating at least $\log N$ steps. We improve this work of oblivious deterministic simulation by a factor of $\log N$ when the memory is clear, and by a factor of $\log\log N$ when the memory is contaminated.

In the rest of the paper, we present the model in Section 2, contamination-tolerant algorithms are in Section 3, we cover general simulations and algorithm transformations in Section 4.

2 Model and definitions

The basis of our model is the restartable fail-stop CRCW PRAM that is discussed and justified by Buss et al. in [4], except that the shared memory that does not contain the input is *contaminated*:

1. There are P PRAM processors. Each has a unique processor identifier PID $\in \{0, \ldots, P-1\}$.
2. *Shared* memory is accessible to all processors; each processor has a constant size *private* memory. Each memory cell stores one word of size $O(\log \max\{N, P\})$.
3. The input is stored in N cells in shared memory.
4. The shared memory not containing the input is *contaminated*.

To enable algorithm termination and sensible accounting of resources, the work of the processors is structured using *update cycles.* Each cycle consists of reading a small number of shared memory cells, performing a fixed time computation, and writing a small number of shared memory cells. The number of reads and writes per cycle is fixed, but depend on the instruction set of the PRAM. The *fail-stop with restart* failure model is defined as follows:

1. A failure pattern F (i.e., failures and restarts) is determined by an *on-line adversary*, that knows everything about the algorithm and is unknown to the algorithm.
2. Any processor may fail at any time in any update cycle, and it may later restart, provided:
   (i) at any time at least one processor is executing an update cycle that successfully completes;
   (ii) single bit writes are *atomic*, i.e., failures can occur before or after a write of a single bit.
3. Failures do not affect the shared memory, but the failed processors lose their private memory. Processors are restarted at their initial state with their PID as their only knowledge.

Condition 2(i) makes termination possible. Update cycles also serve as units of accounting. They do not constrain the instruction set of the PRAM, however the processors are not charged for the instructions of the update cycles that are not completed. (In the absence of update cycle accounting, a *thrashing* adversary can force quadratic work for any *Write-All* solution [4].)

A *failure pattern* F is specified as a set of triples $<$*tag*, PID, $t>$ where *tag* is either **failure** for a processor failure, or **restart** for a restart, PID is the processor identifier, and t is the time when the processor either stops or restarts. The *size* of F is defined as the cardinality $|F|$.

The complexity measure *completed work* generalizes the *Parallel-time*$\times$*Processors* product:

Definition 2.1 Consider an algorithm with P initial processors that terminates in parallel-time τ after completing its task on some input data I of size $|I| = N$, and in the presence of any pattern F of failures and restarts of size $|F| \leq M$. If $P_i(I,F) \leq P$ is the number of processors completing an update cycle at time i, and c is the time required to complete one update cycle, then we define *completed work* as: $S = S_{N,M,P} = \max_{I,F}\{c\sum_{i=1}^{\tau} P_i(I,F)\}$. □

Remark 1 The incomplete work cycles are not counted in S. When the restarts do not occur, then the maximum work spent in the incomplete cycles is bounded by $O(P)$, since there can be no more than P failures. Therefore, for the fail-stop no-restart model, using completed work S yields the same results as using the *available processor steps* measure in [11].

We use the notation "*Write-All*(N,P,L)" to stand for an instance of fault-tolerant *Write-All* that uses P processors and *clear* auxiliary memory of size L to initialize to 1 an array of size N.

Definition 2.2 An algorithm that uses P processors to solve a *Write-All* problem of size N is *contamination-tolerant*, if it is a *Write-All*$(N,P,0)$ algorithm. □

3 Write-All algorithms

The *Write-All* algorithms and simulations based on *Write-All* paradigm, e.g., [11, 12, 13, 20], or the algorithms that can serve as *Write-All* solution, e.g., the addition algorithm in [5] or the maximum finding algorithm in [18], invariably assume that a linear portion of shared memory is either cleared or is initialized to known values. Starting with a non-contaminated portion of memory, such algorithms and simulations are able to perform their computation by "using up" the clear memory, and concurrently or subsequently clearing additional segments of memory needed for future iterations. We develop an efficient *Write-All* solution that requires no clear shared memory.

3.1 A Bootstrap procedure

We formulate a *bootstrap* approach to the design of fault-tolerant *Write-All* algorithms, such that the auxiliary memory is initially contaminated. The bootstrapping procedes in stages:

In stage 1 of our procedure, all P processors clear an initial segment of N_0 locations in the auxiliary memory.

At the stage i of the procedure, we use P processors to clear N_{i+1} memory locations with the help of N_i memory locations that were cleared in the stge $i-1$.

If $N_{i+1} > N_i$ and $N_0 \geq 1$, then this procedure will clear the required N memory location in at most N stages. Say τ is the final stage number, i.e., $N_\tau = N$.

Let P_i be the number of active processors that initiate phase i, and define $N_{-1} = 0$. The cost of such a procedure is: $S_{boot} = \sum_{i=1}^{\tau} S_i(N_i, P_i, N_{i-1})$ where S_i is the cost of the *Write-All*(N_i, P_i, N_{i-1}) algorithm used in stage i.

The efficiency of the resulting algorithm depends on the choices of the particular *Write-All* solution(s) used in each stage and the parameters N_i.

One specific approach is to define a series of multipliers G_0, G_1, ... , G_τ such that $N_i = \prod_{j=0}^{i} G_j$. The high level view of such algorithm is given in Figure 1. The algorithm consists of an initialization (lines 02-04) and a parallel loop (lines 04-09). We use a variation of this scheme below.

We next use the bootstrap approach to construct and analyze contamination-tolerant *Write-All* algorithms in the fail-stop and restartable fail-stop models.

3.2 Algorithm Z for the fail-stop model

We use algorithm W of Kanellakis and Shvartsman [11] and its analysis by Martel [16]. We call algorithm Z the algorithm that results from using W in each phase of the bootstrap procedure.

We analyze algorithm Z for the following choice of parameters: we use $G_0 = \log N$, and $G_i = G_{i-1} \log N$ (for $i > 0$). In the initialization, all P processors traverse a list of size G_0 sequentially and clear it. Then, iteratively, the processors use algorithm W to clear increasingly

```
01  forall processors PID=0..$P-1$ parbegin --Use $P$ processors to clear $N$ memory
02      Clear the initial block of $N_0 = G_0$ elements sequentially using $P$ processors
03      $i := 0$ --Iteration counter
04      while $N_i < N$ do
05          Use a Write-All solution with data structures of size $N_i$
06          and $G_{i+1}$ elements at the leaves
07          to clear memory of size $N_{i+1} = N_i \cdot G_{i+1}$
08          $i := i + 1$
09      od
10  parend
```

Figure 1: A high level view of the bootstrap algorithm.

larger sections of memory using the auxiliary memory cleared in the previous iteration (Fig. 1, lines 05-07).

Algorithm W is a fail-stop (no restart) *Write-All* solution. It uses two full binary trees (represented as heaps in memory) and it consists of a loop in which the active processors synchronously iterate through the following phases:

W1: enumerate the processors in a bottom-up traversal of the processor tree;

W2: allocate the processors in a divide-and-conquer top-down traversal of the progress tree;

W3: work at the leaves; and

W4: evaluate progress in a bottom-up traversal of the progress tree.

To avoid a complete restatement, the reader is urged to refer to [11]. Martel showed the following upper bound for algorithm W:

Theorem 3.1 [16] Algorithm W with P processors, the progress tree with H leaves ($P \leq H$) and $2H - 1$ total nodes all initialized to zero and G array elements at each leaf, has the work of $S = O((H + P \log H / \log\log H) \cdot (\log P + \log H + G))$ for any pattern of stop-failures.

Note that the above result and algorithm W can be used when $P > H$. As described in [4], when there are P processors and the progress tree has $H < P$ leaves, then it is sufficient for each processor to take its PID modulo H to assure uniform initial assignment of processors and to preserve the result.

Algorithm W stores its binary trees as linear arrays interpreted as heaps. Therefore the structure of the trees is unaffected by the state of the memory, because the heaps are imlicit. We next observe that the enumeration of the processors in phase W1 of algorithm W can be done in a bottom-up traversal of a *contaminated* processor tree. The pseudocode for this algorithm is given in Figure 2. We call it algorithm Z_{enum}. The surviving processors enumerate themselves using a standard logarithmic time algorithm based on addition. The contaminated memory cells are distinguished from the cells that contain valid values via the use of a single bit associated with each cell (a so called "deadman flag"). When a processor arrives at a node,

```
01  forall processors $PID = 0..P-1$ parbegin
02      shared integer array $c[1..2N-1]$; --processor counts
03      shared bit array $alive[1..2N-1]$; --alive/dead markers
04      private integer $pn$ --enumerated processor number
05      private integer $j1, j2$, --left/right siblings indices
06          $t$; --predecessor index of j1 and j2
07      $j1 := PID + (N-1)$; --heap-leaf init
08      $pn := 1$; --assume this processor is no. 1
09      $c[j1] := 1$; --a processor is counted once in this step
10      for $1..\log(P)$ do --traverse the tree from leaf to root
11          $t := j1$ div 2; --parent of j1 and j2
12          if $2 * t = j1$
13          then $j2 := j1 + 1$ --j1 came from left
14          else $j2 := j1 - 1$ --j1 came from right
15          fi ;
16          $alive[j2] := 0$ --mark siblings dead
17          $alive[j1] := 1$ --mark self alive
18          if $alive[j2] = 1$ --both sub-trees have active processors?
19          then $c[t] := c[j1] + c[j2]$ --both branches are active
20              if $j1 > j2$ --j1 came from right, update processor n umber
21              then $pn := pn + c[j2]$
22              fi
23          else $c[t] := c[j1]$ --all siblings failed
24          fi ;
25          $j1 := t$ --advance up the heap
26      od
27  parend
```

Figure 2: Contamination robust processor enumeration Z_{enum}.

it clears the bit associated with its sibling, then it sets its own bit (lines 16-17). Only cells that have valid values written in them by active processors will have the bit set. The enumeration itself is as in phase W1.

Theorem 3.2 Algorithm Z is a contamination-tolerant *Write-All*$(N, P, 0)$ algorithm that fo any pattern of fail-stop errors has $S = O(N + P\log^3 N/(\log\log N)^2)$ for $1 \leq P \leq N$.

Proof: We first evaluate and then total the work of the algorithm during each of the finite numbers stages of its execution. In each use of algorithm W, we will have $G = \log N$ as the number of memory locations associated with each leaf of the progress tree, and we will apply Theorem 3.1 with different instantiations of H to evaluate the upper bound of work.

Stage 0: Enumerate processors using Z_{enum}, then sequentially clear $\log N$ memory using all surviving processors. The work using the initial $P_0 \leq P$ processors is: $W_0 = P_0 \cdot \log P + P_0 \cdot \log N$.

Stage 1: $P_1 \leq P_0 \leq P$. Using instance of Theorem 3.1 where $H = \log N$, the work is:

$$W_1 = (\log N + P_1 \log\log N / \log\log\log N) \cdot (\log P_1 + \log N + \log\log N).$$

Stage i: $P_i \le P_{i-1} \le N$. Using instance where $H = \log^i N$:

$W_i = (\log^i N + P_i \cdot i \log\log N/(\log i + \log\log\log N)) \cdot (\log P_i + \log N + i \log\log N)$

The *Final Stage* τ is when $\log^\tau N = N/\log N$, i.e., $\tau = \frac{\log N}{\log\log N} - 1$.

Totalling the work in all phases yields:

$$S = \sum_{i=0}^{\tau} W_i = W_0 + \sum_{i=1}^{\tau} \left(\log^i N + P_i \frac{i \log\log N}{\log i + \log\log\log N}\right) (\log P_i + \log N + i \log\log N)$$

Simplifying the sum results in $S = O(N + P\log^3 N/(\log\log N)^2)$. □

This approach has the following range of optimality:

Theorem 3.3 Algorithm Z is a contamination-tolerant *Write-All*$(N, N(\log\log N)^2/\log^3 N, 0)$ algorithm with $S = O(N)$ for any pattern of fail-stop errors.

3.3 Algorithm Z_r for the restartable fail-stop model

Algorithm Z_r is similar to algorithm Z, except that in each stage we will be utilizing a restartable *Write-All* algorithm. (Algorithm W that is not suitable when restarts are allowed, see [4]). Other parameters of the bootstrap procedure are the same as for the fail-stop case.

In this analysis, we will be using an algorithm that was described and characterized with the following result by Anderson and Woll:

Theorem 3.4 [1] There exists a *Write-All*(H, H, H) solution with H processors that has work $O(H^{1+\varepsilon})$ for every $\varepsilon > 0$.

This is an existential result, and we call this algorithm AW. The best known constructed deterministic algorithm has $\varepsilon = \log_2 3 - 1 < 0.59$ as was shown by Buss et al. [4] (algorithm X, that can also be used with the bootstrap). Note that algorithm AW was developed for the asynchronous model, but it can be used in the restartable fail-stop model as well. The work of the algorithm in the asynchronous model is the same as its completed work in the restartable fail-stop model.

Theorem 3.5 Algorithm Z_r is a contamination-tolerant *Write-All*$(N, N, 0)$ algorithm that fo any pattern of fail-stop errors has $S = O(N^{1+\varepsilon})$ for any $\varepsilon > 0$.

Proof: We first note that there exists a *Write-All*(H, P, H) solution with $P \ge H$ processors that has work $O(P^{1+\varepsilon})$ for every $\varepsilon > 0$. We use algorithm AW, except all processors use their PIDs modulo H. The worst case work is achieved when up to $\lceil \frac{P}{H} \rceil$ processors that have the same PID module H operate synchronously as a single processor. The work of the algorithm in this case is: $S = \lceil \frac{P}{H} \rceil \cdot O(H^{1+\varepsilon}) = O(P^{1+\varepsilon})$. Using this algorithm at each stage of the bootstrap procedure, and evaluating the total work as in Theorem 3.2 yields the desired result:

We evaluate and then sum the work of the algorithm during each of the finite numbers stages of its execution. In each stage $i > 1$ of algorithm Z_τ, we will use algorithm AW $\log N$ times to clear $\log^{i+1} N$ memory locations. In each instance of use of Theorem 3.4, we will use $\delta > 0$ as the exponent, such that $\varepsilon/2 = \delta$. This is done to simplify the final sum using the property that $\log N = O(N^\delta)$ for any $\delta > C$ We also use $P = N$ for clarity.

Stage 0: All processors linearly initialize the segment of shared memory of length $\log N$ using The work is: $W_0 = P \cdot \log N$.

Stage 1: The algorithm is applied $\log N$ times to clear a segment of shared memory of size $\log^2 N$. Using instance where $H = \log N$, the work is: $W_1 = (P \log^\delta N) \cdot \log N$.

Stage i: Using instance $H = \log^i N$: $W_i = (P(\log^i N)^\delta N) \cdot \log N = (P \log^{i\delta} N) \cdot \log N$.

Final Stage τ where $\log^\tau N = N/\log N$, i.e., $\tau = \log N/\log\log N - 1$. Using the instance where $H = \log^\tau N = N/\log N$, the work is: $W_\tau = (P(\log^\tau N)^\delta) \cdot \log N = (P(N/\log N)^\delta) \cdot \log N = P \cdot N^\delta \log^{1-\delta} N$.

$$S = \sum_{i=0}^{\tau} W_i = W_0 + \sum_{i=1}^{\tau} (P \log^{i\delta} N) \cdot \log N = O(N^{1+\delta} \log^{1-\delta} N) = O(N^{1+\delta} \log N) = O(N^{1+\varepsilon}).$$

□

4 Algorithm simulations and algorithm transformations

4.1 Oblivious simulations

Using general simulation techniques [12, 20], if $S_w(N, P)$ is the efficiency of solving a *Write-All* instance of size N using P processors, and if a linear amount of clear memory is available, then a single N-processor PRAM step can be deterministically simulated using P fail-stop processors and work $S_w(N, P)$. Thus if the *Parallel-time* × *Processors* of an original N-processor algorithm is $\tau \cdot N$, then the work S of the fault-tolerant version of the algorithm will be $O(\tau \cdot S_w(N, P))$.

For the setting with initially contaminated shared memory, using algorithms Z and Z_τ with the simulation techniques [12, 20], we obtain the following results:

Theorem 4.1 Any N-processor, τ parallel time PRAM algorithm can be simulated using $O(N)$ contaminated memory and P fail-stop CRCW processors with $S = O(P \log^3 N/(\log\log N)^2 + \tau \cdot N + \tau \cdot P \log^2 N/\log\log N)$ for $1 \le P \le N$.

This simulation has optimal ranges:

Corollary 4.2 Any N-processor, τ parallel time PRAM algorithm can be simulated using $O(N)$ contaminated memory and P fail-stop CRCW processors with $S = O(\tau \cdot N)$ when:

(1) $1 \le P \le N(\log\log N)^2/\log^3 N)$, or

(2) $1 \le P \le N \log\log N/\log^2 N)$ and $\tau > \log N/\log\log N$.

In the restartable fail-stop model we get:

Theorem 4.3 Any N-processor, τ parallel time PRAM algorithm can be simulated using $O(N)$ contaminated memory and N restartable fail-stop CRCW processors with $S = O((1+\tau)\cdot N^{1+\epsilon})$.

Remark 2 Buss et al. [4] define an amortized complexity measure of ***overhead ratio*** σ that measures the computational overhead of an algorithm relative to the necessary work and the number of failures that are encountered. The simulation in the restartable fail-stop model has overhead ratio per PRAM step of $\sigma = N^{\epsilon}$. This overhead ratio can be made polylogarithmic by interleaving algorithm Z_r with algorithm V as presented in [4].

4.2 Improving oblivious simulations

In addition to serving as the basis for oblivious simulations, any solution for the *Write-All* problem can also be readily used as a building block for custom transformations of efficient parallel algorithms into robust ones [11]. Custom transformations are interesting because in some cases it is possible to improve on the work of the naïve oblivious simulation. These improvements are most significant for fast algorithms when a full range of processors is used, i.e., when N are used to simulate N processors, because in this case the parallel slack cannot be taken advantage of. For example in the models with clear initial memory, a factor of $\log N/\log\log N$ was saved off the pointer doubling simulations [11], and using randomization and off-line adversaries, improvements can be obtained in expected work of other algorithms [17, 18].

We next show how to obtain determinsitic savings in work for the prefix sums algorithm that occurs in solutions of several important problems [3]. Efficient parallel algorithms and circuits for computing prefix sums were given by Ladner and Fischer in [14], where the ***prefix problem*** is defined as follows: Given an associative operation $\oplus$ on a domain $\mathcal{D}$, and $x_1, \ldots, x_n \in \mathcal{D}$, compute, for each k, $(1 \le k \le n)$ the sum $\bigoplus_{i=1}^{k} x_i$.

Prefix sums can be computing robustly by using a naïve simulation of a standard logarithmic time algorithm. When using $P = N$ processors, the work of such simulation will be $O(S_w \cdot \log N)$.

Prior to dealing with prefix sums, we make a simple observation that improves on another general simulation. It follows from the fact that since algorithms W and AW, by their definition implement tree traversals, they can be used to implement an associative operation on N values:

Theorem 4.4 Given an associative operation $\oplus$, and an array $x[1..N]$, then $\bigoplus_{i=1}^{N} x[i]$ can be computed using N fail-stop processors at a cost of a single application of algorithms Z (or Z_r).

This saves a full $\log N$ factor over oblivious simulations. We extend Theorem 4.4 and show a robust prefix sum algorithm whose work complexity is $O(S_w)$. In the no-restart fail-stop model we have the following result:

Lemma 4.5 Parallel prefix for N values can be computed using N non-restartable fail-stop processors using $O(N)$ clear memory with $S = O(N \log^2 N / \log\log N)$.

Proof: The prefix summation algorithm that we are going to use as the basis, is an iterative version of the recursive algorithm of [14]. The algorithm consists of two stages: (1) a binary summation tree is computed, and (2) each prefix sum is computed from the summation tree obtained in the first stage, each prefix sum requires no more than logarithmic number of additions.

Each of the two stages can be performed in logarithmic time in parallel by up to N processors. To produce the robust version of the above algorithm, we implement the above stages using the controls of algorithm W with appropriate modifications as follows:

1. In the first stage, a binary summation tree is computed in bottom up traversals at the same time when the progress tree of algorithm W is being updated. This modification to the algorithm does not affect its asymptotic complexity.
2. In the second and final stage, the work phase of algorithm W is modified to include the logarithmic time summation operations using the summation tree as input (as in Theorem 4.4).

   This stage is shown in Figure 3. In the code, $\langle\langle i \rangle\rangle$ is a binary string representing the value i in binary, where most significant bit is bit number 0, and [illegible]$_h$ is the true/false value of the h^{th} most significant bit of $\langle\langle i \rangle\rangle$.

   The loop in lines 09-18 is the top-down traversal of the summation tree. In lines 13-17 the appropriate subtree sum is added (line 14) at depth h only if the corresponding bit value of the processor PID is *true*.

Therefore the work to compute prefix sums is the same as the worst case work of algorithm W. □

Thus we have realized a multiplicative factor of $\log N$ savings over the oblivious simulation when the memory is clear.

Note that because of the lower bounds shown by Beame and Hastad [2] and Li and Yesha [15], at least $\log N / \log\log N$ parallel time and at least $N \log N / \log\log N$ work will be required by $P = N$ processors to compute the prefix sums in the absence of failures. Therefore the multiplicative overhead in work of our parallel prefix algorithm is only $\log N$ when using algorithm W in the fail-stop model.

Using Lemma 4.5 we obtain the following result when the memory is contaminated:

Theorem 4.6 Parallel prefix for N values can be computed using N fail-stop processors and $O(N)$ contaminated memory with $S = O(N \log^3 N / (\log\log N)^2)$.

Note that using N processors to simulate a parallel prefix would require the work (Theorem 4.1) $S = O(N \log^3 N / \log\log N)$, and so the custom algorithm saves a $\log\log N$ factor relative to the oblivious simulation.

```
forall processors $PID = 0..N$ parbegin
    shared integer array $sum[1..2N-1]$; --summation tree
    shared integer array $prefix[1..N]$; --prefix sums
    private integer $j, j1, j2$, --current/left/right indices
            $h$; --depth in the summation tree
    $j := 1$; $h := 0$; --begin at the root, and at depth 0
    $prefix[PID] := 0$; --initialize the sum
    while $h \neq 0$ do --traverse from root to leaf
        $h := h + 1$; $j1 := 2 * j$; $j2 := j1 + 1$ --left/right indices at a new depth
        if $\langle\langle PID \rangle\rangle_h$ --Is the sub-sum at this level included?
        then $prefix[PID] := prefix[PID] + sum[j1]$ --add the left sub-sum
            $j := j2$ --go down to the right
        else $j := j1$ --go down to the left
        fi ;
    od
parend
```

Figure 3: Second stage of contamination-tolerant prefix computation.

Acknowledgements:

I thank Paris Kanellakis for quality feedback that de-contaminated the contents of this report.

References

[1] R. Anderson and H. Woll, "Wait-Free Parallel Algorithms for the Union-Find Problem", *Proc. of the 23rd ACM Symposium on Theory of Computing*, pp. 370-380, 1991.

[2] P. Beame, J. Hastad, "Optimal bounds for decision problems on the CRCW PRAM," *Journal of the ACM*, vol. 36, no. 3, pp. 643-670, 1989.

[3] G. Bilardi and F. P. Preparata, "Size-Time Complexity of Boolean Networks for Prefix Computation," *Journal of the ACM*, vol. 36, no. 2, pp. 363-382, 1989.

[4] J. Buss, P.C. Kanellakis, P. Ragde, A.A. Shvartsman, "Parallel algorithms with processor failures and delays", Brown Univ. TR CS-91-54, 1991. (Prel. version appears as P. C. Kanellakis and A. A. Shvartsman, "Efficient Parallel Algorithms On Restartable Fail-Stop Processors", in *Proc. of the 10th ACM Symp. on Principles of Distributed Computing*, pp. 23-36, 1991.)

[5] R. Cole and O. Zajicek, "The APRAM: Incorporating Asynchrony into the PRAM Model," in *Proc. of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pp. 170-178, 1989.

[6] R. Cole and O. Zajicek, "The Expected Advantage of Asynchrony," in *Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures*, pp. 85-94, 1990.

[7] E. W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Comm. of the ACM*, Vol. 17, No. 11, pp. 643-644, 1976.

[8] S. Fortune and J. Wyllie, "Parallelism in Random Access Machines", *Proc. the 10th ACM Symposium on Theory of Computing*, pp. 114-118, 1978.

[9] P. Gibbons, "A More Practical PRAM Model," in *Proc. of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pp. 158-168, 1989.

[10] M. P. Herlihy, "Impossibility Results for Asynchronous PRAM", in *Proc. of the Third ACM Symposium on Parallel Algorithms and Architectures*, pp. 327-336, 1991.

[11] P. C. Kanellakis and A. A. Shvartsman, "Efficient Parallel Algorithms Can Be Made Robust", to appear in *Distributed Computing*, vol. 5, no 4; prel. version appears in *Proc. of the 8th ACM Symposium on Principles of Distributed Computing*, pp. 211-222, 1989.

[12] Z. M. Kedem, K. V. Palem, and P. Spirakis, "Efficient Robust Parallel Computations," in *Proc. 22nd ACM Symposium on Theory of Computing*, pp. 138-148, 1990.

[13] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. Spirakis, "Combining Tentative and Definite Executions for Dependable Parallel Computing," in *Proc 23d ACM STOC*, pp. 381-390, 1991.

[14] L. E. Ladner, M. J. Fischer, "Parallel Prefix Computation", *Journal of the ACM*, vol. 27, no. 4, pp. 831-838, 1980.

[15] M. Li and Y. Yesha, "New Lower Bounds for Parallel Computation," *Journal of the ACM*, vol. 36, no. 2, pp. 671-680, 1989.

[16] C. Martel, personal communication, March, 1991.

[17] C. Martel, A. Park, and R. Subramonian, "Work-optimal Asynchronous Algorithms for Shared Memory Parallel Computers," to appear in *SIAM Journal on Computing* in 1992.

[18] C. Martel, R. Subramonian, and A. Park, "Asynchronous PRAMs are (Almost) as Good as Synchronous PRAMs," in *Proc. 32d IEEE Symp. on Foundations of Computer Science*, pp. 590-599, 1990.

[19] N. Nishimura, "Asynchronous Shared Memory Parallel Computation," in *Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures*, pp. 76-84, 1990.

[20] A. A. Shvartsman, "Achieving Optimal CRCW PRAM Fault-Tolerance", in *Information Processing Letters*, vol. 39, pp. 59-66, 1991.